 ISSN: 3139-0862

CivicGuide AI: An Intelligent Government Services Information Assistant Using Rule-Based Query Processing

1st Maseeha Banu
Assistant Professor Dept. Of CSE(AI&ML)
Malnad College of Engineering
Hassan, India
mb@mcehassan.ac.in

2nd C R Pruthvik
B.E, Dept. of CSE (AI&ML)
Malnad College of Engineering
Hassan,india
crpruthvikravi@gmail.com

3rd Teju R S
B.E, Dept. of CSE (AI&ML)
Malnad College of Engineering
Hassan,india
tejurs12312@gmail.com

4th D Samiksha Jain
B.E, Dept. of CSE (AI&ML)
Malnad College of Engineering
Hassan,india
samikshajaind@gmail.com

5th Deekshitha K P
B.E, Dept. of CSE (AI&ML)
Malnad College of Engineering
Hassan, India
deeksha.kpgowda006@gmail.com

6th Mohammed Arshil
B.E., Dept. of CSE (AI&ML)
Malnad College of Engineering
Hassan, India
mohammedarshil313@gmail.com

Abstract*—Citizens often face difficulty locating clear information about government services, welfare schemes, scholarships, eligibility conditions and required documents because the information is distributed across multiple government portals. This paper presents CivicGuide AI, a webbased Indian civil-services information portal with a bilingual citizen-assistance chatbot. The running application is implemented using Python and FastAPI. Its chat endpoint accepts a user message and language preference, then passes the query to a dedicated processing engine. The engine uses text normalization, keyword and phrase matching, intent-specific conditions and curated structured datasets to generate responses about scholarships, caste certificates, voter services, civic services, examinations, welfare schemes and state-specific information. The portal also provides dedicated information pages, search and an eligibility-checking feature. The implementation examined here is rule-based and data-driven; it does not use the separate RAG pipeline, ChromaDB, sentence embeddings or Cohere generation service present elsewhere in the repository. This paper describes the application launched by the project entry point, its architecture, query-processing workflow, functionality and limitations. Initial manual functional checks using multiple questions, including specific servicerelated queries, produced relevant responses in the cases tested. A larger controlled evaluation with a documented test set and measured response times remains necessary.

***Keywords*—E-governance, government services, citizen assistant, rule-based chatbot, intent matching, FastAPI, bilingual information system, web application**

I. INTRODUCTION

Access to government services depends on people knowing which scheme applies to them, whether they are eligible, which documents are required and where to apply. In practice this information is spread across many department websites, notices and circulars, and is often written in formal language that is hard for ordinary users to follow. As a result, many citizens depend on intermediaries or search engines that return long lists of links instead of direct answers.

Conversational systems can provide a more accessible way to ask questions than navigating multiple pages. Some modern assistants use large language models and retrievalaugmented generation (RAG) to formulate answers from external documents [1]–[4]. However, not every civic information assistant requires a generative model. A rulebased system can map recognizable user phrases to predefined information handlers and curated data. For a bounded set of services, this approach offers predictable response logic and avoids depending on a model to invent procedural details. Its limitations include sensitivity to wording, incomplete pattern coverage and the need to maintain the underlying information.

This paper describes the application that is started by running python CivicguideAI.py. The launcher runs the FastAPI application app.main:app. The /api/chat endpoint passes the submitted message and language preference to process_citizen_query() in app/data/chatbot_engine.py. The engine uses keyword and phrase matching with structured service datasets to select a response. The repository also contains a separate RAG-oriented implementation under src/, but that is not the application started by this launcher and is therefore outside the implementation described in this paper.

II. LITERATURE SURVEY

A. Conversational Assistants and Query Processing

Conversational assistants range from fixed-response systems to data-driven and generative systems. Transformerbased language models such as BERT and GPT have expanded the capabilities of natural-language processing [1], [5], [6]. Instruction-tuned models can follow user requests [7], while research on hallucination highlights the risk of unsupported generated content [2]. These approaches provide useful background, but the running CivicGuide application described here does not call a large language model. It uses deterministic matching rules and predefined response content.

B. Rule-Based and Data-Driven Civic Information

A rule-based assistant identifies relevant terms or phrases in a user query and routes the query to a corresponding handler. Structured datasets can hold service descriptions, eligibility information, portal links and other domain-specific details. This design is straightforward to inspect and update, but the coverage of the system depends on the rules and data maintained by its developers. Retrieval-augmented generation systems [3], [4], [9], [10] represent a different design choice: they retrieve passages from a document collection and use a generative model to formulate responses. CivicGuide's active launcher follows the rule-based design rather than that RAG workflow.

C. Conversational Assistants for Public Services

Government service portals commonly organize information through menus, forms, search facilities and service-specific pages. CivicGuide combines these portalstyle information pages with a chat interface. The chatbot selects predefined response handlers based on matched words and phrases, while the rest of the portal exposes structured information and interactive tools such as scholarship filtering and eligibility checking. This design is intended to make common information easier to locate; it does not guarantee that every possible wording or question will be understood.

III. SYSTEM ARCHITECTURE

The application launched by CivicguideAI.py consists of a browser-based frontend, a FastAPI backend, a chat endpoint, a rule-based query-processing engine and structured datasets containing information about government services. The backend also exposes routes for portal pages, search and eligibility

 ISSN: 3139-0862

checking. The chatbot engine imports datasets for scholarships, caste certificates, voter services, civic services, state portals, examinations, welfare schemes and frequently asked questions.

For a chat request, the browser sends a message and a language value to the POST /api/chat endpoint. The endpoint extracts these fields and calls process_citizen_query(message lang). The query-processing engine normalizes and examines the text, detects relevant words or phrases using matching helper functions, and evaluates topic-specific conditions. When a supported intent is identified, the engine returns a predefined response containing a title, answer text, an official-source field and suggested links. The endpoint returns this response as JSON to the frontend. The application also provides separate routes for pages such as scholarships, voter services, civic services, state directory and FAQs, along with search and eligibility endpoints.

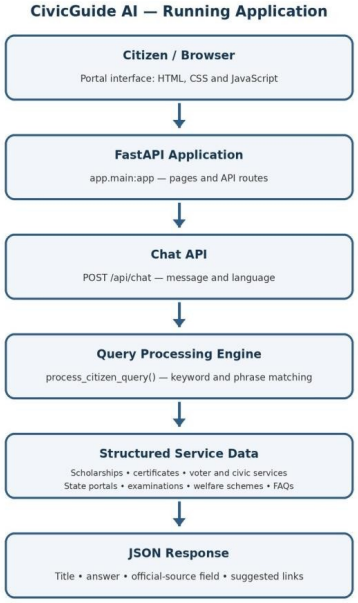


Fig. 1. Architecture of the running CivicGuide AI portal launched by CivicguideAI.py.

IV. METHODOLOGY

A. Knowledge Base Preparation

The chatbot knowledge content is maintained in Python data modules imported by app/main.py and app/data/chatbot_engine.py. These modules contain structured records for scholarships, caste certificates, voter services, civic services, state portals, examinations, welfare schemes and FAQs. The query engine uses helper functions such as has_word() and has_any() to check whether a query contains configured words or phrases. The system does not perform document chunking, vector embedding or ChromaDB indexing in this active application path.

B. Query Matching

Let q denote the normalized user query and P = {p1, p2, …, pm} denote the set of configured patterns or phrases. The query engine checks for matches between q and the patterns associated with supported topics. A matching topic activates its corresponding response handler, which selects information from the relevant structured dataset or predefined response text.

The matching function returns 1 when a configured phrase from a topic pattern set occurs in the query; otherwise, it returns 0. The selected handler returns a structured response containing fields such as title, answer, official_source and suggested_links. If no supported pattern is identified, the engine returns a default response that lists the categories it can assist with. This matching approach is deterministic for a given query and dataset version, but it can fail when users use unfamiliar wording or ask questions outside the supported topics.

C. Response Construction

The system does not generate an answer using a probabilistic language model. Instead, the query engine selects a predefined response branch based on detected words, phrases, language selection and topic-specific conditions. The response is assembled from text and structured data maintained in the application.

The /api/chat endpoint returns the response generated by process_citizen_query(). The chatbot engine supports English and Hindi response paths through its language handling. Since responses are selected from programmed rules and data, the system behaviour can be inspected in code. However, correctness still depends on the quality, completeness and currency of the maintained information.

D. Context and Chat History

The chat endpoint processes the message and language value supplied in each request. The frontend may display conversation messages as part of its interface, but the inspected backend call to process_citizen_query() receives the current message and language; no previous-turn transcript is passed into that function. Therefore, the application should not be described as using conversational memory to interpret follow-up questions.

V. IMPLEMENTATION

The running portal is implemented in Python with FastAPI. CivicguideAI.py starts Uvicorn with app.main:app on port 8000. The main application is defined in app/main.py where routes render the portal pages and expose APIs including /api/chat, /api/search and /api/check-full-eligibility. The /api/chat route calls process_citizen_query() from app/data/chatbot_engine.py. The chatbot engine imports domain datasets from the app/data package and uses keyword and phrase matching to select response logic. The frontend uses HTML, CSS and JavaScript with static assets and Jinja2 templates.

TABLE I. TECHNOLOGY STACK OF CIVICGUIDE

Layer	Technology	Role
Application entry point	Python, Uvicorn	CivicguideAI.py starts app.main:app on port 8000
Backend	FastAPI, Jinja2	Provides web pages and APIs including /api/chat, /api/search and eligibility checking
Chat processing	Python rule-based engine	Matches words and phrases and selects topic-specific response handlers
Knowledge content	Python structured data modules	Stores service, scholarship, welfare, exam and portal information used by handlers
Frontend	HTML, CSS, JavaScript	Presents portal pages and sends chat requests to /api/chat

The frontend submits the user message and language preference to /api/chat and displays the JSON response. The portal also includes dedicated pages for scholarships, caste certificates, voter services, civic services, state information and FAQs. The /api/search route searches the structured datasets, while /api/check-full-eligibility applies programmed conditions to user-provided details and returns matching scholarship and service information. The launcher binds the server to 0.0.0.0:8000 and opens the local browser at http://127.0.0.1:8000.

VI. RESULTS AND DISCUSSION

A. Functional Validation

The running CivicGuide AI application was manually tested by submitting multiple questions through its chat interface. The test queries included specific questions about the information handled by the portal, rather than only broad or general prompts. For the questions tested, the chatbot returned relevant answers that were observed to match the corresponding information

 ISSN: 3139-0862

maintained in the application data and response rules. This provides initial functional evidence that the keyword- and phrase-matching engine can identify supported topics and select appropriate predefined responses, including for more specific queries. The observed behaviour is consistent with the intended workflow: the submitted message is processed by process_citizen_query(), a matching response handler is selected, and the resulting answer and related fields are returned to the frontend.

These checks indicate that the chatbot functions for the questions exercised during manual testing. However, the test set, exact number of questions, expected answers and individual pass/fail outcomes were formally recorded.

B. Response Time

The launcher starts the server locally on port 8000. Although the application responded to the questions used during manual functional checks, response time was not measured systematically. A performance evaluation should record the test machine, network conditions, question set, warm-up procedure and repeated measurements. Accordingly no average latency or performance comparison is claimed in this paper.

C. Comparative Analysis

Table II provides a qualitative comparison of broad system-design characteristics. It is a conceptual comparison, not the result of a controlled experiment.

TABLE II. COMPARISON WITH EXISTING APPROACHES

Feature	General-purpose chatbot	Government portal	CivicGuide AI
Interaction	Natural-language conversation	Menus, keyword search or forms	Portal pages plus a rule-based chat interface
Answer source	Depends on model and configuration	Published portal content	Predefined response logic and structured application datasets
Response method	May use generative language models	Displays pages and service workflows	Keyword/phrase matching selects programmed response handlers
Updating information	May require prompt, model or knowledge updates	Portal owner updates content	Developers update the Python data modules and response rules
Follow-up handling	Depends on conversational context support	Usually page or form based	Current chat request is processed without prior-turn transcript

D. Limitations

The chatbot can respond only to topics and wording covered by its programmed matching rules and datasets. Queries phrased in an unexpected way may fail to activate the intended handler or may return the default response. Information can become outdated unless the underlying data modules are reviewed and updated. The rule-based engine does not understand arbitrary follow-up questions through conversation history, and it does not use semantic vector retrieval or generative language-model reasoning. Users should confirm important eligibility, deadlines and procedures through the relevant official government portal.

VII. FUTURE SCOPE

Future work may include a structured evaluation of intent coverage and response correctness using a larger, documented set of questions and verified expected answers; expanding and periodically reviewing the structured government-service datasets; improving phrase matching for spelling variations and natural language; adding broader regional-language support; and evaluating whether a retrieval-based or generative component would provide measurable benefits. Any such component should be described as a future extension unless it is actually integrated into the running application.

VIII.CONCLUSION

This paper presented CivicGuide AI, a web-based government-services information portal with a bilingual, rule-based citizen-assistance chatbot. The application launched by CivicguideAI.py runs app.main:app and processes chat requests through /api/chat and process_citizen_query(). Its responses are selected using keyword and phrase matching, programmed conditions and curated structured datasets. The portal also provides dedicated service pages, search and eligibility-checking functionality. Initial manual checks with multiple general and specific questions produced relevant responses in the cases tested; however, a formal benchmark and response-time evaluation remain future work. The repository contains a separate RAG implementation, but it is not part of the application launched by the command examined in this paper. Future work will focus on evaluating response correctness, broadening query coverage and maintaining current service information.

ACKNOWLEDGMENT

The authors thank the Department of Computer Science and Engineering (AI & ML), Malnad College of Engineering, Hassan, for providing the necessary infrastructure, facilities, and academic support required to carry out this mini project. The authors also express their sincere gratitude to their guide and faculty members for their valuable guidance, encouragement, and support throughout the development of CivicGuide AI: Intelligent Government Services Information Assistant.

REFERENCES

[1] T. B. Brown et al., “Language models are few-shot learners,” in Proc. Adv. Neural Inf. Process. Syst. (NeurIPS), 2020, pp. 1877–1901.

[2] Z. Ji et al., “Survey of hallucination in natural language generation,” ACM Comput. Surv., vol. 55, no. 12, pp. 1–38, 2023.

[3] P. Lewis et al., “Retrieval-augmented generation for knowledgeintensive NLP tasks,” in Proc. NeurIPS, 2020, pp. 9459–9474.

[4] Y. Gao et al., “Retrieval-augmented generation for large language models: A survey,” arXiv preprint arXiv:2312.10997, 2023.

[5] A. Vaswani et al., “Attention is all you need,” in Proc. NeurIPS, 2017, pp. 5998–6008.

[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pretraining of deep bidirectional transformers for language understanding,” in Proc. NAACL-HLT, 2019, pp. 4171–4186.

[7] L. Ouyang et al., “Training language models to follow instructions with human feedback,” in Proc. NeurIPS, 2022, pp. 27730–27744.

[8] H. Touvron et al., “LLaMA: Open and efficient foundation language models,” arXiv preprint arXiv:2302.13971, 2023.

[9] V. Karpukhin et al., “Dense passage retrieval for open-domain question answering,” in Proc. EMNLP, 2020, pp. 6769–6781.

[10] K. Guu, K. Lee, Z. Tung, P. Pasupat, and M.-W. Chang, “REALM: Retrieval-augmented language model pre-training,” in Proc. ICML, 2020, pp. 3929–3938.

[11] N. Reimers and I. Gurevych, “Sentence-BERT: Sentence embeddings using Siamese BERT-networks,” in Proc. EMNLP-IJCNLP, 2019, pp. 3982–3992.

[12] J. Johnson, M. Douze, and H. Jégou, “Billion-scale similarity search with GPUs,” IEEE Trans. Big Data, vol. 7, no. 3, pp. 535–547, 2021.